THIS PAGE BLANK (USPTO)

PCT/ SE 00 / 0 1 4 9 4

# PRV
PATENT- OCH REGISTRERINGSVERKET
Patentavdelningen

10/031332

REC'D 23 AUG 2000
WIPO PCT

**Intyg**
**Certificate**

*Härmed intygas att bifogade kopior överensstämmer med de handlingar som ursprungligen ingivits till Patent- och registreringsverket i nedannämnda ansökan.*

*This is to certify that the annexed is a true copy of the documents as originally filed with the Patent- and Registration Office in connection with the following patent application.*

SE 00/01494

(71) *Sökande* — *Telefonaktiebolaget L M Ericsson, Stockholm SE*
*Applicant (s)*

(21) *Patentansökningsnummer* — *9902752-6*
*Patent application number*

(86) *Ingivningsdatum* — *1999-07-21*
*Date of filing*

4

*Stockholm, 2000-08-15*

*För Patent- och registreringsverket*
*For the Patent- and Registration Office*

Åsa Dahlberg
*Åsa Dahlberg*

*Avgift*
*Fee*

**PRIORITY DOCUMENT**
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

Telefonaktiebolaget L M Ericsson

A processor architecture

This invention relates to a processor architecture of the kind disclosed in the preamble of claim 1. The processor is particularly, but not exclusively, adapted to execution of functional programs.

TECHNICAL FIELD OF THE INVENTION

Program development is very costly. These costs could sometimes be reduced if the program could be made in a functional language since it is hard for a user to use a machine programmed in a conventional language - it requires a lot of knowledge - and because of the complexity a programmer might introduce hidden errors.

The development of programming languages

The development of the first electronic computer started the development of several programming languages suited for this type of computer, such as FORTRAN, COBOL, Algol, BASIC, Pascal. These languages have been called imperative languages, below also called conventional languages, mainly because of the fact that they normally give programs that consist of a sequence of commands or instructions to be executed sequentially by a conventional computer, i.e. a computer designed according to the principles developed by John von Neumann. Imperative kind of programs have became increasingly complex and often contain a lot of errors, are difficult to read, difficult to understand and particularly hard to modify.

An increasing discomfort with these imperative languages led to the development of another series of languages, so called functional languages: LISP, ISWIM, Scheme (a dialect of LISP), ML, Hope, SASL, and so on. The driving force behind the development of these languages was conceptual simplicity; no particular machine influenced the design. Functional programming languages have several properties alleviating some of the disadvantages of the more conventional programming languages.

For additional information and understanding we refer to the textbook "Functional Programming Using Standard ML", Åke Wikström, Prentice Hall 1987.

A program written in a functional language can be seen as a set of definitions of properties of objects and as computation rules. The definitions are the declarative part and the computation rules are the operational part that the computer uses during execution. Functional languages provide a higher-level interface to the computer, which makes it possible for the programmer to abstract away from hardware-related details of the computer. As a positive side-effect, functional programs are often shorter and easier to understand than conventional imperative programs.

Nowadays functional languages are implemented as a virtual machine on a conventional processor. This has been done with compilers or interpretating programs. This means that a program is executed which interprets the program instructions. Every program instruction results in that a number of machine instructions are executed. The execution will therefore be slow.

It is clear that some of the benefits of the functional program approach have been held back by the fact that practically no useful dedicated hardware has been on the market for the process of storing and executing functional programs in an effective manner. Some processors adapted to execute functional programming languages, so called FFP machines, are discussed in the textbook "High-level Language Computer Architecture" (ISBN 0,88175-1342-4) from 1988, chapters 11 and 12, and have been constructed and manufactured from time to time. Dedicated Lisp working stations were for instance furnished to the middle of the eighties. A transputer from Inmos Ltd, which is a processor for built-in systems was introduced in the middle of the eighties and is still on the market.

A particular kind of functional language is Erlang, which is developed by the Applicant for real-time applications. In spite of the disadvantage that functional programs are slow to execute on the von Neuman kind of computer, much slower than the imperative languages, the use of Erlang is increasing. However, the capacity of the computers having Erlang installed is not enough for some applications. Erlang programs are built as a number of communicating processes, also called processes. A switch is often made between execution between different processes. This switch is improductive and should be made as fast as possible. For the moment being this is made in software, in a so called "run-time system".

Since data programs written in functional languages, such as Erlang, provided on conventional computers are very power consuming, too much for a lot of applications, there is a need for providing a hardware having a low power dissipation. Low power dissipation is essential in most products. Demand for low power dissipation will be increasing in the future. High power dissipation hinders further processor speed improvement.

## SUMMARY

### OBJECTS OF THE INVENTION

An object of the invention is to provide a micro processor which is adapted to execute programs or at least parts of programs written in a language having a sequential instruction flow, for instance in Erlang.

Another object of the invention is to provide a co-processor executing a functional language in a real-time operative system. One or several co-processors of this kind should be able to handle processes demanding great capacity.

Still another object of the invention is to provide a processor, particularly dedicated to Erlang.

Yet another object of the invention is to provide a processor adapted to a functional language in wireless portable equipment. Processors in wireless portable equipment are provided with a content of software (SW) which often need be changed. In such a case a processor adapted to handle a functional language, such as Erlang, should give a great advantage.

Still another object of the invention is to provide a processor adapted to a functional language having fewer memory accesses than what is common today. This means a need for a high instruction density.

Another object of the invention is to provide a device able to handle much functionality implemented in software and which has a low or reasonable power dissipation, thus to provide a great amount of functionality to low power consumption.

Yet another object of the invention is to provide dedicated hardware support for garbage collection and process switching in the processor in order to improve execution performance and lower power dissipation. Such mechanisms are commonly implemented in a "run time system" in software.

INVENTION

The invention relates to a processor architecture adapted to program languages operating with a sequential instruction flow and handling data through use of lists or tuples or simple types and comprising an instruction holding means, a data memory means storing data objects, and execution means. The objects mentioned above can be solved by providing means for handling references to data objects referenced by bindings and comprising means to increment reference counts to a data object and to decrement reference counts to a data object in dependence of an actual instruction from the instruction holding means.

Storage means could be provided in the means for handling temporary storage of data, and to keep notice of bindings to said temporary storage. When the notice keeping means detects generation of a zero reference to an object, meaning that this object is not needed anymore, it preferably makes the memory slot for that object available as a free memory slot. The means for handling temporary storage of data comprises preferably a parameter memory means having means for keeping notice of the bindings to the stored values and having means for storage of said data values.

Value storage means could store values and type information for the values fed to the parameter memory. The values, being a part of the data objects, have then the bindings to the parameters. The parameter memory will then transfer values be-

tween functions using the parameters and using the parameters for temporal storage. The parameter memory could replace parameter references in fetched instructions from the instruction memory means with stored actual values before computation. Means could be provided in the parameter memory for storing and managing environment information for the parameters, where the environment determines which parameters are currently valid parameters. Means could be provided in the parameter memory for storing and managing information for parameters and environments, where process information fed to the parameter memory is used to determine which environments and which parameters are currently valid.

A process identification register could be provided for identification of the currently executed process, and an environment identification register for identification of the currently executed environment. At least the top of at least one priority queue of processes to be executed is preferably kept available for reading. At least part of the process descriptor of the next to be executed process is then kept available for reading in the parameter memory means. In order to make a process switch:

* a new environment is created in the parameter memory, and at least the program counter is stored in said new environment,
* said new environment value is stored in the process descriptor of the current process, said process descriptor may be stored in the data memory,
* the environment value of the new process is restored,
* the new process is set to be the current process,
* at least the program counter is restored.

The instructions are preferably provided with only one instruction format, where each instruction is composed of a distinct number of sub-instructions. Each sub-instruction has in turn the same and only one instruction format comprising a first part and a second part, the first part determining the action to take and the second part providing a value to use in the action.

The invention could be adapted to execution of functional languages. Then a set of instructions are created comprising dedicated instructions for function calls, function returns, parameter transfer between functions. Then also a set of instructions could be created comprising dedicated instructions for incrementing and decrementing of memory references. The processor architecture disclosed above could be adapted to process parts of computer programs written in a functional language.

## ADVANTAGES

The processor architecture is designed to perform particularly well for the following cases:

- "Functional application" which constitutes the controlling structure in a functional language.

- Process switching, below called context switching (Other expressions for processes are tasks or threads. The expression process(es) will be used in the remaining of the text, although it should be understood that tasks or threads could be used as well). Normally, Telecom applications are implemented using a large number of processes. It is essential to be able to switch between these processes fast, in order not to loose performance.
- Message passing between processes.
- Memory management, including garbage collection.

Functional languages leave memory management to the run-time system. Handling memory takes time from the processor and is generally regarded as a problem. When using the invention provided with for instance Erlang, this is done in the processor by a dedicated unit and interfering very little with the useful computing.

The processor according to the invention executes the instructions in a RISC like manner, i.e. in a sequential instruction flow easy to pipeline. Thus, the processor according to the invention is not a reduction machine. It has a simple combinatory instruction decoding, efficient execution for function calls, support for creation and manipulation of lists and tuples, support for fast context switching, and high code density. This keeps the power dissipation on a low and economical level avoiding the need for costly cooling systems.

## BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention and for further objects and advantages thereof, reference is now made to the following description taken in conjunction with the accompanying drawings, in which:

FIG 1 illustrates a first embodiment of a dedicated processor architecture for handling functional languages working with a sequential instruction flow, such as Erlang.

FIG 2 illustrates a preferred embodiment of an instruction pipeline.

FIG 3 illustrates an embodiment of the internal structure of the parameter memory included in the processor according to the invention.

FIG 4 illustrates the hierarchy in the specification of the parameters stored in the parameter memory.

## DETAILED DESCRIPTION OF EMBODIMENT

The instruction set architecture (ISA) in the processor according to the invention is designed to be compact and efficient for the execution of functional languages and particularly for Erlang. However, the inventive concept is also adaptable to some other kinds of languages than functional. Some design goals for the ISA are identified below:

- Execution in a RISC like manner, i.e. a sequential instruction flow easy to pipeline.
- Simple instruction decoding.
- Efficient execution for function calls.
- Support of fast context switching.
- High code density.
- Supporting garbage collection for lists an tuples.

There is preferably only one instruction format. Every instruction is preferably composed of a distinct number of sub-instructions, for example three, each having the same format. The first sub-instruction format comprises a first part, which determines the action to take, and a second part, which provides a value used in the action taken. The first sub-instruction may determine the interpretation of the full instruction, or may allow other sub-instructions to instruct concurrent independent actions in a Very Long Instruction Word (VLIW) manner.

Reference is now made to FIG 1, which illustrates an embodiment of the main architecture of the processor according to the invention, which fulfils the features mentioned above. The program for the processor is written into an instruction memory 2. The program is stepped forwards by a program counter 1 providing an address to the instruction memory 2.

As illustrated in FIG 2, the processor architecture is in this embodiment basically a four stage pipeline, comprising the stages Fetch IF, Instruction Decode ID, Parameter Memory Access PMA and Execute EX, as will be further described below. Since stores can be pipelined a fifth store stage also could be regarded as a part of the pipeline.

Returning to FIG 1, the instruction to be fetched is pointed out by the program counter 1. It is fetched from the instruction memory 2 by the instruction fetch mechanism IF. Therefore, the program counter is connected to the instruction memory 2 and feeds an address to the memory. The memory then returns the instruction pointed to by the address.

The fetched instruction is stored in the memory 2 in coded form, i.e. in a form compiled into the machine code of the processor, and is then fed to an instruction decoder 3, which makes the Instruction Decode ID. Unconditional branches are de-

tected during the decoding stage. The decoder 3 transfers the instruction into a number of control signals functioning as control words.

The decoded instruction is then transferred into a parameter memory 4 stage, for the Parameter Memory Access PMA. This parameter memory 4 is a hardware means of a particular kind provided for this invention. Within this stage parameter references in the decoded instruction will be substituted by their corresponding actual values which are stored in the parameter memory 4, as will be apparent below in relation to FIG 3.

Thus the processor does not operate with registers or a stack. It is instead operating with parameters. The parameters are temporary bindings to values, which are temporarily stored in the parameter memory 4 in the processor. Parameters are allocated according to need and are freed upon completion of function application or upon explicit instruction to be. Parameters are accessed according to context. The maximum number of parameters is not limited by the architecture but depends on the specialised processor implementation. This gives a good and efficient support for handling of function arguments and local variables/bindings.

Parameters thus take a central part in the instruction set architecture of the processor, according to the invention, as will be illustrated below. Parameters are used for arguments of functions and local bindings made within a function body, and for argument transfer at function calls.

The parameter memory 4 is designed to provide a fast instruction execution and stores current bindings, i.e. function parameters and local variables. This means that parameter transfer, local bindings and function results do not go through the main data memory 5 and instead through the fast parameter memory 4.

Within this parameter memory stage certain decoded instructions will create new parameter bindings by storing actual values in the parameter memory 4 and creating parameter bindings to these values. Stored values may be provided from the decoded instruction or may be provided from a register, or may be fetched from a data memory 5. Several operations of either kind of store and substitute may be done during each clock cycle.

During the parameter memory access stage parameter bindings in an instruction are replaced with actual values. The parameter memory 4 could preferably support at least two such replacements per clock cycle. After the parameter memory stage parameter bindings in an instruction have been substituted with actual values where the actual values are fetched from the parameter memory.

The data memory handler 6 keeps record of the addresses in the data memory 5, because in functional languages, such as Erlang, the program is not working with addresses and pointers as in imperative languages. Such features are hidden in a so called run-time procedure which means that the data memory handler 6 is needed to

determine and keep record of the addresses for the particular registers comprising the data for the processes stored in the data memory 5. Addresses for stored data is delivered back to the execution unit 7.

The parameter memory 4 feeds substituted instruction data and code words to the execution unit 7, which executes the instruction. The parameter memory 4 comprises also a register into which at least the current identity, id, of a process is stored, as will be discussed further when describing FIG 3.

The parameter memory 4 is in fact the unit, which reads the registers in the data memory 5, while an execution unit 7 connected to the parameter memory 4 is the unit, which writes the data into the registers. The data memory handler 6 is connected to co-operate with both the parameter memory 4 and the execution unit 7.

Similarly to the language Erlang the processor architecture is based upon processes. A first dedicated register in the parameter memory holds the current process id, *cpr*. A second dedicated register holds the current environment identity, ***env***. An environment means a sequence of instruction within which certain parameter references are valid, for instance the scope of the value bindings.

The preferred structure of the parameter memory 4 is presented in FIG 3. It comprises a process storage plane 10 in which the current process, process_info, is stored, an environment storage plane 11 in which the information of the environments, environment_info, for the process is stored.

While not directly visible in the instruction set, processes are used to keep track of processes (or tasks or threads) in the high level language executed on the processor. There is a correspondence between an Erlang process and a process in the processor. A spawn instruction creates a new process, initiates its environment and associates it with its process descriptor. Another instruction pushes a process on a process queue. Another instruction switches out the current process and switches in the first process in the process queue to become the new current process.

At least the following information is stored in the process descriptor:

- Pointer to first message.
- Pointer to last message.
- Current environment (when not executing).
- Other misc information, such as a list of linked processes.

Also, the architecture is based upon the concept of environments. An environment (*env*) defines the current bindings in the parameter memory 4. Environments are used to keep track of parameter scope. A new environment, *env*, is created at the beginning of a function call and becomes the current environment. The parameter bonds in a function call are bonded in the new environment. At a function return the current environment is terminated, and the environment of the calling function is restored as the current environment, i.e. the current environment is replaced with the

previous environment. All parameter bindings in the replaced environment are purged at function return.

As illustrated in FIG. 4, a parameter is valid only within its environment. Similarly an environment is valid only within its process.

Thus, when an instruction is provided which makes a function call then *env* is stepped up with 1. When an instruction comes, which indicates a jump back then *env* is stepped down with 1.

The parameter memory 4 also has a storage plane 12, for instance comprising a register *crp*, holding process identity for a certain parameter value and type stored in plane 13. As illustrated in FIG 3, searching for the parameter id, actual environment, *env*, and actual process is transmitted to the value storage 13.

The parameter memory 4 can perform a number of actions. Examples of these actions and their corresponding results and required inputs are listed below.

Read: The parameter memory 4 returns the type and value, stored in the value storage plane 13, of the specified parameter. Information of the parameter id, actual environment and actual process is provided to the storage planes 10, 11 and 12 in the parameter memory 4.

Pop: The parameter memory 4 returns the value + type (stored in the storage plane 13) of the specified parameter. In addition it eliminates the parameter and its value from the parameter memory 4. Information of the parameter id, actual environment and actual process is provided to the storage planes 10, 11, and 12 in the parameter memory 4.

Set: The parameter memory 4 stores a new parameter with a specified id (in the type and value storage plane 13). Information of the parameter id, actual environment, actual process and parameter value is provided to the planes 12, 11, and 10 in the parameter memory 4.

Garb: The parameter memory eliminates the parameter and its value from the planes 10-13 in the memory. Information of the parameter id, actual environment and actual process is provided to the parameter memory 4.

Garb env: The Parameter Memory eliminates all parameters and their values from the planes 10-13 for the specified environment value; information of the environment and the actual process is provided to the parameter memory 4.

A read operation could, for example, work in the following way. The three leftmost storage planes 10, 11, 12 in FIG 3 perform an associative search. The line yielding hits in the three storage planes selects the value + type data stored for that line in the rightmost storage plane 13 in FIG 3.

The function is similar for storing, except that value + type data is stored instead of being read. The garb and pop operations discard the information in the addressed position and make it free to use.

Thus, the parameter memory 4 is used to store values, and type information for these values. Parameters are bound to the values where the parameters are used to transfer values between functions and are used for temporal storage. The parameter memory 4 can replace parameter references in fetched instructions with stored actual values before computation.

Search in the files could be made in some kind of associative process. However, the search could be implemented in some other way as well, for example to use compromised addresses by for example process and *env*. It is also possible to connect the process storage to an associative memory and the environment storage 11. Then, the parameter reference is fed to the parameter memory and makes a search. A value is out-putted.

The parameter memory 4 can also manage environment information for the parameters where the environment determines which ones of the stored parameters, which are the currently valid parameters.

The parameter memory 4 could as well manage process information for the parameters and environments where the process information determines which ones of the stored environments, which are currently valid environments, and which of the stored parameters, which are currently valid parameters.

This means that parameters are used instead of registers or stack. This leads to more efficient handling of function calls and local bindings. A context dependent mechanism is used for addressing the parameters. This makes the instruction set architecture here described independent of the amount of parameter storage in a particular processor implementation.

At least one process queue is administrated by the processor. Several queues may be administrated, for instance for different priority levels.

Since the Parameter Memory 4 has a limited number of parameter slots it may not be sufficient to hold all parameters for a large program at the same time.

If the Parameter Memory 4 begins to reach its capacity limit, parameters begin to be stored in the data memory 5, thus freeing slots in the Parameter Memory 4. This action is called swap out of parameters. The opposite action, called parameter swap in, reads previously stored parameters from the data memory 5 and restores them in the Parameter Memory 4 with correct process, environment and parameter id values.

Both parameter swap in and swap out are done automatically by the processor without interfering the instruction execution.

The top of the process queue(s) is observed. If a process (PF) is found there which is not currently in the Parameter Memory 4, parameters for a process (which is not the current process) in the Parameter Memory 4 become swapped out, and parameters in the topmost environments of the found process (PF) are swapped in to the Parameter Memory.

A certain mechanism determines which process, which should have it s parameters swapped out. For instance, Least Recently Used (LRU) could be the strategy by which it is decided which process to swap out.

If for the current process the number of environments in the Parameter Memory begins to reach the maximum limit, an activity is started to swap out parameters of the lowest environment to the data memory. Similarly, if the number of environments in the Parameter Memory begins to reach the lower limit, an activity is started to swap in parameters of the highest environment stored in the data memory (if any).

The parameter swap in and swap out mechanisms ensures that the most likely to be used parameters are stored in the Parameter Memory 4.

Regarding the memory management, the memory slots in the data memory 5 not used are organised as a list of available memory registers. A determined register (free) in the data memory handler 6 points to the first element of that list. Upon a request for storage of an element, this element is stored in the memory slot denoted by the mentioned dedicated register (free). The register (free) is then updated to point to the now actual first available memory position, below called memory slot, which earlier was the second in the list. Upon releasing a used memory slot this slot is placed in the beginning of the list of available memory slots.

As mentioned above Erlang handles data through the use of tuples and lists and simple types. These data are referenced by bindings. When a data object is not referenced by a binding, it is not needed any more and can be thrown away. Its memory slot can then be used for other data objects. Thus, memory slots to which no reference is made are considered free to use. This means that an automatic garbage collection could be regarded as being made. Notice is kept of such memory slots, and they are released upon request.

Memory management is thus supported well in this architecture and is performed according to the principle of reference counting. This means that some sub-instructions control incremented reference and decremented reference, respectively, for data objects. When such a sub-instruction is executed the reference part of a data object is either incremented or decremented automatically by the unit for memory management. If a zero reference occurs and is detected its memory slot is made free and available as free memory slot for usage of other data objects.

As mentioned above, the architecture of the parameter memory 4 is based on processes and there will often be a need to switch from one process to another process. Then the following operations need be done.

Upon a context switch the current environment identity is stored in a Process (or Task) Descriptor pointed out by the process register (cpr). Other registers may be stored in the parameter memory if needed. When the process is switched back the register values can then be restored from the parameter memory. Thus, the only access to the data memory 5 needed to accomplish a context switch is to store and restore the environment.

In order to make a switch:

- Bind data values of registers in the current working set to parameters and store them in the parameter memory 4. The value of the current environment is stored in the process descriptor of the current process. *Env* then functions as a key for pointing to where in the parameter memory 4 a reading should begin at restore.
- Read *env* and restore into the parameter memory 4 the new "working set" of the new process to which the switch is made from the process descriptor in the data memory 5 comprising the environment for that new process.
- Restore the program counter 1 to the value next after it was before the actual switched process were switched out.

Following this scheme time consuming memory access to the main memory 5 is minimised, and fast context switching is achieved.

The instruction set architecture could for example be intended for execution of functional languages, supporting the languages with dedicated instructions for function calls, function returns, transfer of arguments between functions, and process management. There are particular instructions for creating, reading, writing and manipulation of the data objects, which occur in Erlang, i.e. lists and tuples and simple types.

<u>Examples of instruction format</u>

*instruction :: = sub-instruction, sub-instruction, sub-instruction*
*sub-instruction :: = tag, value*

where *tag* specifies how *value* should be interpreted. For example: *par, 5* means that parameter number *5* is specified. This presents the logical organisation of the instructions. In an actual processor implementation all tags can for example be grouped together. The tag in the first sub-instruction typically specifies how the full instruction shall be interpreted. If for example *tag* in the first sub-instruction specifies a binary arithmetic operation, the two other sub-instruction specify the operands, and the value of the first sub-instruction specifies the type of arithmetic operation.

There can be exceptions from this rule. Some of the instructions in the first memory slot do only use the first two sub-instructions in which case the third slot can be utilised for a single sub-instruction, i.e. there could be a group of independent sub-instructions which only operates in the third slot in an instruction. Some independent sub-instructions can be placed in the second slot as well.

Examples of tag/value combinations are:

*Tag, value* :: = *fun, fun_number* (call function *fun_number*)
| *par, par_number* (use the specified parameter)
| *pop, register_number* (return back from current function, deliver result in the specified register)
| *reg, register_number* (use the value in *register_number*)
| *alu, alu_operation* (perform the operation given by *alu_operation*)
| *gar, par_number* (free the space used by this parameter)
| etc.

<u>Example</u>

The following registers could be used in the processor:

*acc*: stores the result from alu operations,
*cpr*: stores the current process id,
*env*: stores current environment,
*id*: stores the memory address of the latest stored element,
*res*: holds a result value when returning from a function,
*d0*: holds a data element. Its parts can be addressed by *d01, d02, d03, d04*, where *d01* holds the element type value, *d02* holds a numerical value, *d03* and *d04* holds pointer/integer values including value type,
*free*: stores the first free memory slot.

The processor could use following not addressable registers:
*env-old*: stores the previous environment
*cnt*: stores the last allocated parameter number.

As mentioned above, parameters are used for argument transfer at function calls, and for local bindings. When a function call is executed the arguments are listed in the second and third sub-instructions in the calling instructions. The arguments get consecutive numbers as they appear starting the first sub-instruction in a new function body.

A local binding is made through the *par* or *pad* instructions mentioned below, e.g.
(*pab:5, reg:acc,* .*)

binds parameter 5 to the value in the register *acc* in the current environment in the current process. * represent arbitrary element.

There are also particular instructions for manipulating the references to data, i.e. to support the language with dedicated instructions to increment or decrement the number of memory references to a data object.

Experiences with this ISA (Instruction Set Architecture) has indicated a considerable reduction of instructions in order to make a particular functionality. This means correspondingly fewer numbers of memory accesses. Memory accesses demand much power. A reduction of them will therefore lower the power consumption in the system.

Although the invention is described with respect to exemplary embodiments it should be understood that modifications can be made without departing from the scope thereof. Accordingly, the invention should not be considered to be limited to the described embodiments, but defined only by the following claims, which are intended to embrace all equivalents thereof. As mentioned above, the parameter memory 4 is based on processes. Therefore, this kind of device can be provided even for computers handling other kinds of languages using processes (or tasks or threads) than functional languages, for example C and C++. Context switching is also provided in for instance the language ADA, and thus the features described for the context switching could also be valid for other kinds of languages handling context switching. Data management of the kind described above is also made in for instance the modern language Java and is therefore valid also for such kinds of languages.

We claim

1. A processor architecture adapted to program languages operating with a sequential instruction flow and handling data through use of lists or tuples or simple types, and comprising an instruction holding means (2,3), a data memory means (5) storing data objects, and execution means (7),
**characterized by**:
means (4,5,6) for handling references to data objects referenced by bindings and comprising means (6) to increment reference counts to a data object and to decrement reference counts to a data object in dependence of an actual instruction from the instruction holding means (2,3).

2. A processor architecture according to claim 1, **characterized** by
means (4) for handling temporary storage of data in the means (4,5,6) for handling references to data objects;
storage means (13) in the means (4) for handling temporary storage of data, and to keep notice of bindings to said temporary storage.

3. A processor architecture according to claim 2, **characterized** by
data memory handler means (6) in said means (4,5,6) for handling references to data objects which when detecting a generation of a zero reference to an object, meaning that this object is not needed anymore, makes the memory slot for that object available as a free memory slot.

4. A processor architecture according to any of the preceding claims, **characterized** in
that the means for handling temporary storage of data comprises a parameter memory means (4) having means (10, 11, 12) for keeping notice of the bindings to the stored values, and having storage means (13) for storing said data values.

5. A processor architecture according to claim 4, **characterized** by said storage means (13) in the means (4) for handling temporary storage of data stores values and type information for the values fed to the parameter memory (4), and that the values, being a part of the data objects, have the bindings to the parameters, and that the parameter memory (4) transfers values between functions using the parameters and using the parameters for temporal storage.

6. A processor architecture according to claim 4 or 5, **characterized** in that the parameter memory (4) replaces parameter references in fetched instructions from the instruction memory means (2) with stored actual values before computation.

7. A processor architecture according to any of the claims 4 to 6, **characterized** by means (11) in the parameter memory (4) for storing and managing environment information for the parameters, where the environment determines which parameters are currently valid parameters.

8. A processor architecture according to any of the claims 4 to 7, **characterized** by means (10 - 13) in the parameter memory (4) for storing and managing information for parameters and environments, where process information fed to the parameter memory is used to determine which environments and which parameters are currently valid.

9. A processor architecture according to any of the claims 4 to 8, **characterized** by a process identification register for identification of the currently executed process, and an environment identification register (11) for identification of the currently executed environment.

10. A processor architecture according to any of the claims 4 to 9, **characterized** in that at least the top of at least one priority queue of processes to be executed are kept available for reading, and that at least part of the process descriptor of the next to be executed process are kept available for reading in the parameter memory means (4).

11. A processor architecture according to claims 10, **characterized** in that in order to make a process switch:

* a new environment is created in the parameter memory (4), and at least the program counter is stored in said new environment,
* said new environment value is stored in the process descriptor of the current process, said process descriptor may be stored in the data memory (5),
* the environment value of the new process is restored,
* the new process is set to be the current process,
* at least the program counter is restored.

12. A processor architecture according to any of the preceding claims, **characterized** by instructions having only one instruction format, where each instruction is composed of a distinct number of sub-instructions, each of which has in turn the same and only one instruction format comprising a first part and a second part, the first part determining the action to take and the second part providing a value to use in the action.

13. A processor architecture according to any of the preceding claims adapted to execution of functional languages, **characterized** by a set of instructions comprising dedicated instructions for function calls, function returns, parameter transfer between functions.

14. A processor architecture according to claim 13, **characterized** by a set of instructions comprising dedicated instructions for incrementing and decrementing of memory references.

15. A processor architecture according to any of the preceding claims, **characterized in that** it is adapted to process parts of computer programs written in a functional language.

## ABSTRACT

A processor architecture is adapted to program languages operating with a sequential instruction flow and handling data through use of lists or tuples or simple types. It comprises a program holding means (1), an instruction holding means (2,3), a data memory means (5) storing data objects, and execution means (7). Means (4,5,6) are provided for handling references to data objects referenced by bindings and comprising means (6) to increment reference counts to a data object and to decrement reference counts to a data object in dependence of an actual instruction from the instruction holding means (2,3).

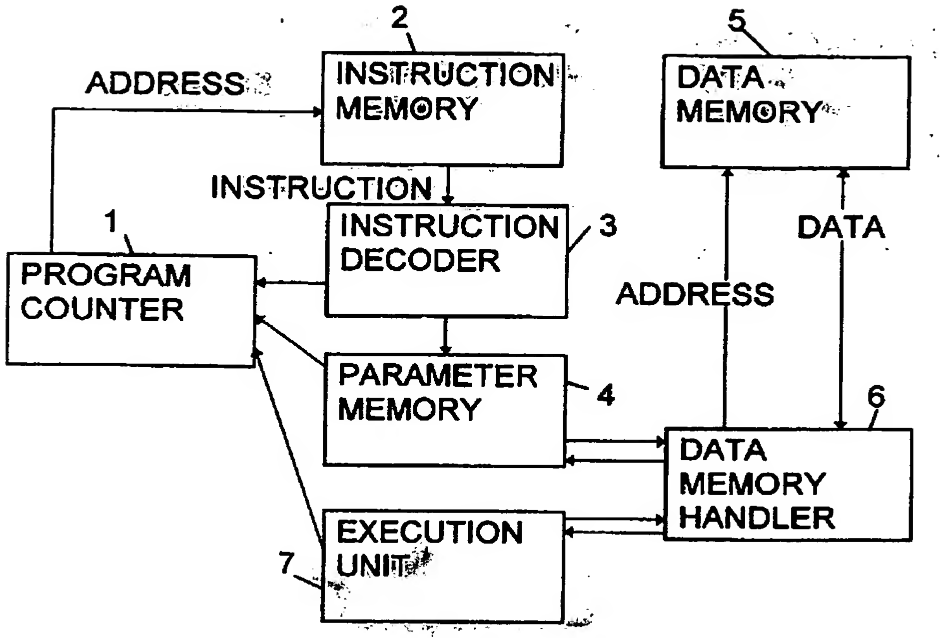
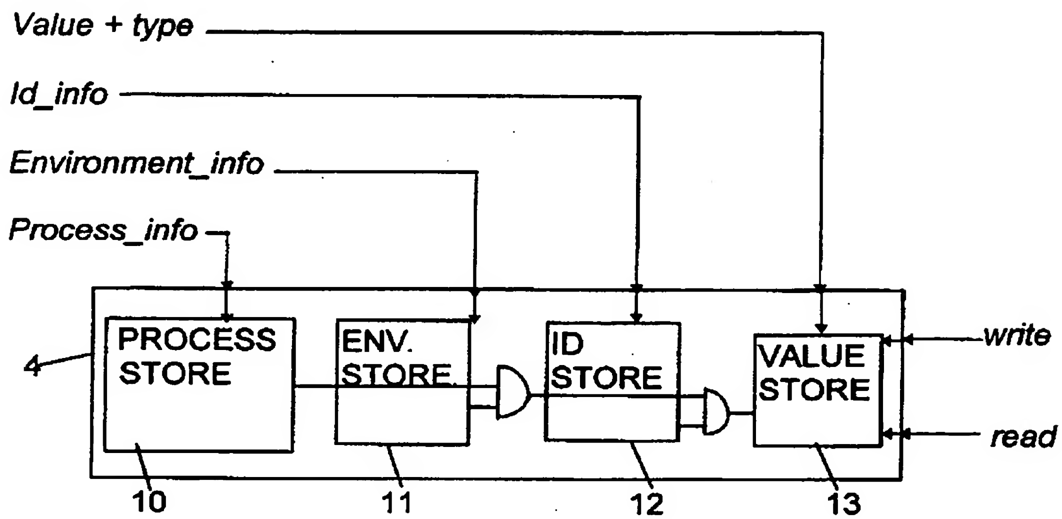



FIG 1

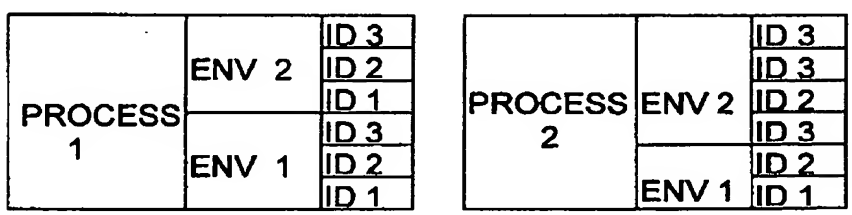
| IF | ID | PMA | EX |
|---|---|---|---|
| INSTRUCTION FETCH | INSTRUCTION DECODE | PARAMETER MEMORY ACCESS | EXECUTE |

FIG 2



Value + type
Id_info
Environment_info
Process_info
4
PROCESS STORE
ENV. STORE
ID STORE
VALUE STORE
write
read
10
11
12
13


FIG 3



PROCESS 1
ENV 2
ID 3
ID 2
ID 1
ENV 1
ID 3
ID 2
ID 1
PROCESS 2
ENV 2
ID 3
ID 3
ID 2
ID 3
ENV 1
ID 2
ID 1


FIG 4

THIS PAGE BLANK (USPTO)

# INTERNATIONAL SEARCH REPORT

International application No.
PCT/SE 00/01494

**A. CLASSIFICATION OF SUBJECT MATTER**

IPC7: G06F 9/44, G06F 12/02
According to International Patent Classification (IPC) or to both national classification and IPC

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)

IPC7: G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

SE,DK,FI,NO classes as above

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | US 4912629 A (ROBERT L. SHULER,JR.), 27 March 1990 (27.03.90), column 1 - column 4; column 6; column 11, line 55 - column 12, line 12 | 1-4,13-15 |
| Y | | 5-8 |
| | -- | |
| Y | US 5555434 A (L. GUNNAR CARLSTEDT), 10 Sept 1996 (10.09.96), column 5, line 56 - column 6, line 6; column 6, line 45 - line 58; column 10, line 10 - column 11, line 47, figure 4, abstract | 5-8 |
| | -- | |
| A | EP 0171934 A2 (TEXAS INSTRUMENTS INCORPORATED), 19 February 1986 (19.02.86), the whole document | 1-8,13-15 |
| | -- | |

[X] Further documents are listed in the continuation of Box C. [X] See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance: the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance: the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

| Date of the actual completion of the international search | Date of mailing of the international search report |
|---|---|
| 1 November 2000 | 09 -11- 2000 |
| Name and mailing address of the ISA/ Swedish Patent Office Box 5055, S-102 42 STOCKHOLM Facsimile No. +46 8 666 02 86 | Authorized officer Erik Veillas/OGU Telephone No. +46 8 782 25 00 |

# INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/SE 00/01494

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| US | 4912629 | A | 27/03/90 | NONE | | | |
| US | 555434 | A5 | 10/09/96 | NONE | | | |
| EP | 0171934 | A2 | 19/02/86 | US | 4695949 | A | 22/09/87 |
| US | 4922414 | A | 01/05/90 | AU | 570657 | B | 24/03/88 |
| | | | | AU | 2217683 | A | 21/06/84 |
| | | | | CA | 1214283 | A | 18/11/86 |
| | | | | CA | 1229682 | C | 24/11/87 |
| | | | | EP | 0113460 | A | 18/07/84 |
| | | | | IL | 70279 | A | 31/07/87 |
| | | | | JP | 59188879 | A | 26/10/84 |
| | | | | US | 4887235 | A | 12/12/89 |
| US | 5535390 | A | 09/07/96 | NONE | | | |
| WO | 9608948 | A2 | 28/03/96 | NONE | | | |